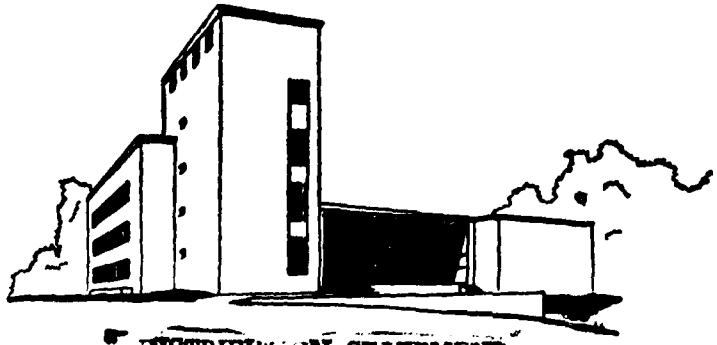
[redacted]

Carnegie Mellon University

PITTSBURGH, PENNSYLVANIA 15213

GRADUATE SCHOOL OF INDUSTRIAL ADMINISTRATION

WILLIAM LARIMER MELLON, FOUNDER



A DYNAMIC SUBGRADIENT-BASED BRANCH AND BOUND PROCEDURE FOR SET COVERING

by

Egon Balas
Carnegie Mellon University

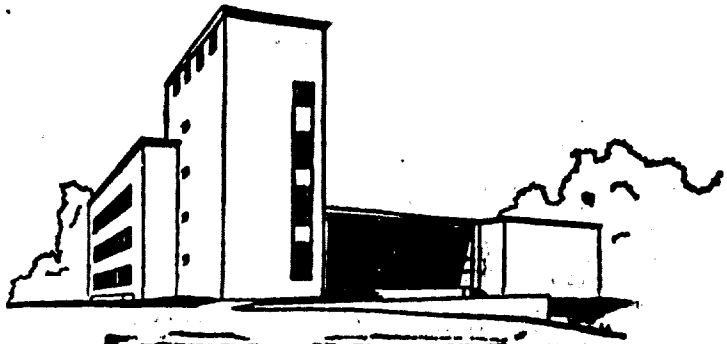
and

Maria C. Carrera
Northeastern University

Carnegie Mellon University

PITTSBURGH, PENNSYLVANIA 15213

GRADUATE SCHOOL OF INDUSTRIAL ADMINISTRATION
WILLIAM LARIMER MELLON, FOUNDER



Management Sciences Research Report No. MSRR 568(R)

A DYNAMIC SUBGRADIENT-BASED BRANCH AND BOUND PROCEDURE FOR SET COVERING

by

Egon Balas
Carnegie Mellon University

and

Maria C. Carrera
Northeastern University

October 1991
Revised May 1992

Management Sciences Research Group
Graduate School of Industrial Administration
Carnegie Mellon University
Pittsburgh, PA 15213

The research of the first author was supported by Grant DDM-8901495 of the National Science Foundation, and Contract N00014-89-J1063 with the U.S. Office of Naval Research. Reproduction in whole or in part is permitted for any purpose of the U.S. Government.

Abstract

We discuss a branch and bound algorithm for set covering, whose centerpiece is an integrated upper bounding/lower bounding procedure called dynamic subgradient optimization (DYNSGRAD), that is applied to a Lagrangean dual problem at every node of the search tree. Extensive experimentation has shown DYNSGRAD to represent a significant improvement over currently used techniques.

1. Introduction

We address the set covering problem

(SC) $\min\{cx : Ax \geq 1,\ x \in \{0,1\}^n\}$,

where A is an m×n matrix of 0's and 1's, c is an integer n-vector, and 1 is the m-vector of 1's. We denote $M := \{1,\ldots,m\}$, $N := \{1,\ldots,n\}$, and $N_i := \{j \in N : a_{ij} = 1\}$, $i \in M$, $M_j := \{i \in M : a_{ij} = 1\}$, $j \in N$.

Using standard terminology, we call the support of a feasible integer solution x to (SC) a *cover*. A cover is called *prime* if it contains no redundant columns. To simplify the terminology, x itself will often be called a cover.

Our approach belongs to a family of branch and bound procedures for (SC) whose main characteristic is that at every node of the search tree, instead of using the simplex method to solve the linear programming relaxation of the given subproblem, it uses some combination of primal and dual heuristics with subgradient optimization applied to a Lagrangean dual, possibly incorporating cutting planes, to generate upper and lower bounds on the objective function value.

Various elements of this approach have been around for about 15 years. Etcheberry [77] seems to have been first to use subgradient optimization instead of the simplex method. Balas and Ho [80] tested various Lagrangean relaxations, some of them including cutting planes, and introduced several primal and dual heuristics combined with variable fixing techniques, as well as some new branching rules. Their algorithm and its computational performance served as a benchmark for subsequent developments in the 80's. Hall and Hochbaum [83] extended the approach to more general covering problems (right hand side greater than one). Vasko and Wilson [84] introduced an efficient randomized version of the greedy heuristic. Carrera [84] performed a thorough computational study of primal heuristics, including the randomized greedy and reduced cost heuristics. Beasley [87] implemented a branch and bound algorithm using the Balas-Ho

framework, but solving the linear programming subproblems by the simplex method and using some new variable fixing rules. In a later paper, Beasley [90] introduced a Lagrangean heuristic that generates a cover after every subgradient iteration. Fisher and Kedia [90] use an efficient dual heuristic as the main ingredient of their approach. For some recent work on set covering algorithms based on other approaches, see Harche and Thompson [89] and Nobili and Sassano [91].

The centerpiece of the approach discussed in this paper is an integrated lower bounding/upper bounding procedure that we call *dynamic subgradient optimization* (DYNSGRAD) and that we apply to a Lagrangean dual problem at every node of the search tree. This procedure intertwines the iterations of subgradient optimization, a lower bounding procedure, with applications of the *reduced cost heuristic* (RCH), a primal, i.e. upper bounding procedure (which however, as a byproduct, also improves the lower bound), followed by variable fixing applied in a recursive fashion. Whenever RCH improves the upper bound, the current dual vector (set of Lagrange multipliers) is changed, along with the upper and lower bounds; and whenever some variable is fixed at 1, the constraint set of the Lagrangean problem itself changes, and the parameters of the subgradient procedure are adjusted: hence the qualifier "dynamic" in the name of the procedure.

Other new features of our algorithm include some primal and dual heuristics, a recursive variable fixing procedure, and new branching rules.

Our procedures have been extensively tested, both on randomly generated and on real world problems. As a stand-alone heuristic, the dynamic subgradient procedure compares favorably with all other heuristics known to us in terms of the quality of solutions obtainable with a certain computational effort. As a new ingredient of our branch and bound procedure, it has drastically improved the performance of the latter as compared to its older version which uses at

every node the standard subgradient optimization procedure. It also compares favorably with other branch and bound procedures.

Our paper is organized as follows. We start by introducing the formulation that we use for our Lagrangean dual (Section 2). Next we briefly review the standard subgradient optimization procedure (SUBGRAD) as it applies to this Lagrangean dual (Section 3). A dual heuristic is described next, for making the vector of Lagrange multipliers into a feasible dual solution without weakening the lower bound that it provides (Section 4). We then review some primal procedures and show why the reduced cost heuristic (RCH) is the best candidate to be used with DYNSGRAD (Section 5). Next we describe our recursive variable fixing procedure (Section 6), and finally state and discuss the dynamic subgradient procedure itself (Section 7). This is followed by a discussion of our two branching rules and the overall branch and bound procedure (Section 8). Computational experience both with DYNSGRAD as a stand-alone, and the dynamic subgradient-based branch and bound procedure (Section 9) concludes the paper.

2. The Lagrangean Dual

The Lagrangean dual to which we apply our procedure is

$$\max_{u \geq 0} L(u)$$

where

$$L(u) := u1 + \min_{x \in \{0,1\}^n} \{(c - u\tilde{A})x : \Sigma(x_j : j \in N_i) \geq 1,\ i \in \bar{M}\}.$$

Here $\bar{M} \subset M$ is the index set of a maximal subset of disjoint (i.e. pairwise nonoverlapping) inequalities of the system $Ax \geq 1$, $\tilde{A}$ is the matrix obtained from A by deleting the rows indexed by $\bar{M}$, and u is a vector indexed by $M \backslash \bar{M}$. For given u^t, we denote by $\mathcal{L}(u^t)$ the minimization problem in x associated with u^t, i.e. the problem that needs to be solved to calculate the value of $L(u^t)$.

This Lagrangean should be contrasted with the simpler one often used in its place, in which all of the constraint set $Ax \geq 1$ is taken into the objective function and the Lagrangean subproblem to be solved at every iteration has no other constraints than $x_j \in \{0,1\}$, $j \in N$. We have tested both versions and although the differences were not drastic, we found the version that we decided to use more reliable and converging somewhat faster.

3. Standard Subgradient Optimization

To establish a framework for discussing our dynamic subgradient procedure, we first describe the standard subgradient optimization procedure (SUBGRAD) as it applies to the above Lagrangean dual of a set covering problem. The j^{th} column of $\tilde{A}$ is denoted $\tilde{a}_j$, and the solution to $\mathcal{L}(u)$ for a fixed u is x(u). We start with upper and lower bounds z_U and z_L on the value of L(u), a vector u, and a set of disjoint constraints indexed by $\bar{M}$. We also need to specify a starting value (taken to be 2) for the parameter λ used to define the step length σ, the number k of iterations with no improvement in the lower bound after which λ is reduced (halved), and values of the parameters used with the stopping rules, namely ε (stop if $\sigma < \varepsilon$), δ (stop if the norm of the subgradient is less than δ) and Ω (stop if the number of iterations exceeds Ω).

This procedure, like others to follow, is stated in mock PASCAL, with comments in braces. Text in italics summarizes steps that don't need further elaboration or will be elaborated later.

procedure *SUBGRAD*
input: *c, A, $\bar{M}$, z_U, z_L, u; λ, k, ε, δ, Ω*
output: {improved} z_L, u^t
begin
    $N^0 := N$, $t := 1$, $u^t := u$ {initialize}
    while $z_U > z_L$ **do** {solve $\mathcal{L}(u^t)$ and define u^{t+1}}
        for $i \in \bar{M}$ **do** {cover optimally the rows in $\bar{M}$}

```
            c_k - u^t ã_k := min{c_j - u^t ã_j : j ε N_i}
            x_k(u^t) := 1, N^0 := N^0\{k}
        endfor
        for j ε N^0 do {assign values to remaining x_j(u^t)}
            if c_j ≤ u^t ã_j then x_j(u^t) := 1
            else x_j(u^t) := 0
        endfor
        L(u^t) := u^t 1 + (c - u^t Ã)x(u^t)
        z_L := max{z_L, L(u^t)}
        for i ε M\M̄ do {compute subgradient}
            g_i(u^t) := 1 - Σ(x_j(u^t) : j ε N_i)
        endfor
        if z_L unchanged for k iterations then
            λ := λ/2
        endif;
        σ^t := λ(z_U - z_L)/‖g(u^t)‖^2 {compute step length σ^t}
        if σ^t < ε or ‖g(u^t)‖ < δ or t > Ω then
            stop
        endif
        for i ε M\M̄ do {update u_i}
            u_i^{t+1} := max{0, u_i^t + σ^t g_i(u^t)}
        endfor;
    endwhile
end.
```

The main advantage of subgradient optimization over the simplex method (or, for that matter, an interior point method), is its low computational cost. It is the common experience of users of this approach that the number of iterations required for convergence does not depend on problem size. What exactly it depends on, besides of course the formulation of the Lagrangean itself, is not well understood. It differs between problem classes, but within the same class there is no evidence of any dependence on problem size. On our rather diverse set of test problems, the number of iterations tended to be around 2-300 and only on very rare occasions did it exceed 400. A different set of stopping

rules may of course change these numbers; in particular, the intervals at which the size of λ is reduced do influence the number of iterations. But this is not the point: rather it is that these numbers do not seem to depend on problem size. Accordingly, after having fine tuned the procedure for a class of problems, i.e. chosen appropriate values for its parameters, one may want to limit the number of iterations, as we did in the above outline, to some constant Ω independent of problem size. With this in mind, and denoting by q the number of nonzero entries of $\tilde{A}$, the complexity of solving the current subproblem $\mathcal{L}(u^t)$ is O(n), that of computing the subgradient is O(q), and that of updating u is O(m); all of which yields a complexity of O(q) for the whole procedure.

4. Dual Heuristic

When the subgradient optimization procedure stops, the dual vector u^t may or may not be feasible (i.e. satisfy $u^tA \leq c$), even though $L(u^t) := u^t1 + (c - u^t\tilde{A})x(u^t)$ is a valid lower bound. Several of the procedures that we will use, however, require as their starting point a feasible dual vector. It is therefore an important fact that there exists a simple procedure for making u^t feasible *without weakening the associated lower bound.*

The procedure for doing this (DUALFEAS) consists of the following. Let $s_j := c_j - u\tilde{a}_j$, $j \in N$. For every column j with $s_j < 0$, decrease the dual variables u_i, $i \in M_j \backslash \bar{M}$, one after the other, until the total decrease equals $-s_j$. Then update all reduced costs to account for the changes in u. This makes all reduced costs nonnegative. To assign a value to the components indexed by $\bar{M}$, for each $i \in \bar{M}$ set u_i equal to the smallest reduced cost over N_i. The details follow.

procedure *DUALFEAS*
input: $u^t \geq 0$ {infeasible}
output: $u \geq 0$ {feasible}

```
begin
    u := u^t, s := c - u^t Ã {initialize}
    for j ε N such that s_j < 0 do
        while s_j < 0 do
            choose i ε M_j\M̄
            if s_j < -u_i then {update s_ℓ, ℓ ε N_i and decrease u_i}
                for ℓ ε N_i do
                    s_ℓ := s_ℓ + u_i
                endfor
                u_i := 0
                M_j := M_j\{i}
            else {decrease last u_i and update s_ℓ, ℓ ε N_i}
                u_i := u_i + s_j
                for ℓ ε N_i do
                    s_ℓ := s_ℓ - s_j
                endfor
        endwhile
    endfor
    for i ε M̄ do {assign values to u_i, i ε M̄}
        s_k := min{s_j : j ε N_i}
        u_i := s_k
    endfor
end.
```

Proposition. *The procedure DUALFEAS produces a vector* $u \geq 0$ *such that* $uA \leq c$ *and*

$$\Sigma(u_i : i \in M) \geq \Sigma(u_i^t : i \in M\backslash\bar{M}) + \Sigma(c_j - u^t\tilde{a}_j)x_j(u^t)(= L(u^t))$$

Proof. Clearly, $u_i \geq 0$, $i \in M$. Since for each j such that $c_j < u^t\tilde{a}_j$ the relevant components of u^t are decreased until $c_j \geq u\tilde{a}_j$, and the components of u for $i \in \bar{M}$ are set so as to preserve these inequalities, u satisfies $s := c - uA \geq 0$. To prove the last inequality of the Proposition, we rewrite it in the form

$$\Sigma(u_i^t - u_i : i \in M\backslash\bar{M}) + \Sigma(s_j^t : s_j^t < 0) \leq \Sigma(u_i : i \in \bar{M}) - \Sigma(\min\{s_j' : j \in N_i\} : i \in \bar{M}),$$

where $s^t := c - u^t\tilde{A}$ and $s' := c - u\tilde{A}$.

We claim that the left hand side of this inequality is nonpositive, while the right hand side is nonnegative. To prove the first claim, we note that every time some u_i^t is decreased by an amount, the same amount gets added to at least one s_j^t such that $s_j^t < 0$. Therefore

$$\Sigma(u_i^t - u_i : i \in M\backslash\bar{M}) \leq -\Sigma(s_j^t : s_j^t < 0).$$

The second claim is true because each u_i, $i \in \bar{M}$, is assigned the value $\min\{s'_j : j \in N_i\}$ during the procedure, and $s'_j \geq s_j$ for all j. □

The complexity of DUALFEAS is easily seen to be $O(q)$.

5. Primal Heuristics

The Dynamic Subgradient Procedure to be introduced below interrupts the standard subgradient procedure at certain iterations and uses a primal heuristic to extend the support of the solution $x(u^t)$ of the current problem $\mathcal{L}(u^t)$ to a cover for (SC). The rationale behind this is that producing "on the fly" a large number of near-optimal covers is a computationally cheap way of increasing the chances of success in the overall search for a good solution. Next we take a brief look at the primal heuristics that could be candidates for use within such a dynamic SGD. We discuss these heuristics as they apply to (SC), i.e. to the original constraint set $Ax \geq 1$ rather than $\tilde{A}x \geq 1$.

The best known primal heuristic is the greedy, which builds a cover sequentially, by adding at every step a column for which some function f of the cost c_j and the number k_j of new rows covered by column j attains its minimum.

The complexity of the greedy heuristic is $O(Cq)$, where C is the cardinality of the cover. In terms of the problem data, this is $O(mq)$. Since in most cases the cardinality of a cover is much smaller than m, the greedy heuristic is fast and so one can run it with several different functions $f(c_j, k_j)$ and choose the best solution obtained. Balas and Ho [80] used five functions as column selection criteria, namely c_j, c_j/k_j, $c_j/\log_2 k_j$, $c_j/k_j\log_2 k_j$ and $c_j/k_j \ln k_j$. Vasko

and Wilson [84] used these same functions plus two others, c_j/k_j^2 and $c_j^{1/2}/k_j^2$. Their randomized greedy heuristic selects the column to be added to the cover at every step by randomly choosing some function $f(c_j,k_j)$ from a specified pool of such functions. The randomized greedy heuristic can be run repeatedly for as long as improved solutions are found. Of course, there is a break-even point between the increasing marginal cost of another run and the decreasing marginal benefit from such a run, as the discovery of improved solutions becomes less and less likely. The experience of Vasko and Wilson [84], confirmed by Carrera [84], was that 30 runs is under most circumstances a reasonable choice.

The computational experience of both Vasko and Wilson [84] and Carrera [84] was that the randomized greedy heuristic, run 30 times, while computationally several times as expensive as running the greedy with each of the functions $f(c_j,k_j)$ and choosing the best solution, performs consistently better than the latter approach.

A heuristic based on a somewhat different philosophy than the greedy, uses information about the reduced costs (in the linear programming sense) provided by a feasible (not necessarily optimal) dual solution, in constructing the (primal) cover. We call a feasible dual solution u maximal, if no component u_i can be increased without decreasing some other component or making u infeasible. The reduced cost heuristic (RCH) introduced by Balas and Ho [80], is based on the simple observation that if the columns with zero reduced cost do not form a cover, then the dual solution is not maximal; in particular, the dual variable associated with an uncovered row can be increased. Such an increase in turn drives to 0 a new reduced cost, and the corresponding column can be added to the cover. Iterating this produces a full cover x, as well as a maximal dual u. Thus as a byproduct to generating a cover, the reduced cost heuristic also improves the dual solution used to start it.

```
procedure RCH
input: c, A, feasible u
output: x {cover}, maximal u
begin
     s := c - uA
     for j ε N do
          if s_j = 0 then x_j := 1 else x_j := 0
     endfor;
     for i ε M do
          if Σ(x_j : j ε N_i) < 1 then
               s_k := min{s_j : j ε N_i}
               u_i := u_i + s_k
               for j ε N_i do
                    s_j := s_j - s_k
                    if s_j = 0 then x_j := 1
               endfor
          endif
     endfor
     reduce x to a prime cover
end.
```

The computational complexity of RCH is O(q), better than that of the greedy heuristic, and in practice RCH is considerably faster than the latter.

As to the quality of solutions, Carrera [84] found the covers generated by RCH better than those obtained by the greedy heuristic with any of the functions $f(c_j, k_j)$ used as selection criterion.

The fact that, along with a (primal) cover, RCH also produces an improved dual solution, eminently qualifies it for use in DYNSGRAD. Further, if we slightly generalize RCH by allowing it to start with some of the reduced costs s_j being negative and replacing the requirement $s_j = 0$ in the first do loop by $s_j \leq 0$, then applying RCH to (SC) with the starting dual vector u defined by $u_i = u_i^t$, $i \in M \setminus \bar{M}$, and $u_i = \min \{s_j : j \in N_i\}$, $i \in \bar{M}$, produces at the end of the first do loop a partial cover identical to the solution $x(u^t)$ of the Lagrangean

subproblem $\mathcal{L}(u^t)$ solved by SUBGRAD. So it is only natural to take this partial cover and complete it by applying RCH starting with the second do loop. This modified/truncated RCH is our *Embedded Reduced Cost Heuristic* (ERCH).

```
procedure ERCH
input: c, A, M̄, u^t, x(u^t), s^t
output: x {cover}, u
begin
    u := u^t, x := x(u^t), s := s^t {initialize}
    for i ε M do
        if i ε M̄ or Σ(x_j : j ε N_i) < 1 then
            s_k := min{s_j : j ε N_i}
            u_i := u_i + s_k
            for j ε N_i do
                s_j := s_j - s_k
                if s_j = 0 then x_j := 1
            endfor
        endif
    endfor
    reduce x to a prime cover
end.
```

As mentioned above, the vector $x(u^t)$ found by SUBGRAD is identical to the partial cover generated by the first do loop of RCH if applied to (SC) with the starting dual vector chosen in a certain way. ERCH completes this partial cover by executing the second do loop of RCH. However, we have found it useful to alternate this procedure with another one that completes the partial cover corresponding to $x(u^t)$ by using a version of the greedy heuristic. We call this the *Embedded Reduced Cost/Greedy Heuristic* (ERCGH).

```
procedure ERCGH
input:  c, A, x(u^t)
output:  x {cover}
begin
```

```
    x := x(u^t) {initialize}
    R := {i ε M : Σ(x_j : j ε N_i) < 1} {identify uncovered rows}
    while R ≠ ø do
        c_k/|R ∩ M_k| := min {c_j/|R ∩ M_j| : j ε N, x_j = 0}
        x_k := 1
        R := R\M_k
    endwhile
    reduce x to prime cover
end
```

At any point in the subgradient procedure, the current dual vector u^t may not be feasible. Whenever ERCH (or ERCGH) finds an improved cover, making u^t feasible by DUALFEAS and then applying RCH to the full constraint set $Ax \geq 1$ is computationally inexpensive and in our experience often yields an improved solution, and leads to additional variable fixing.

6. Recursive Variable Fixing

Whenever ERCH is called to extend a partial solution $x(u^t)$ to a full cover, the result may be a reduction of the integrality gap $z_U - z_L$. If this is the case, a natural followup is to fix at 0 all those variables x_j whose reduced cost s_j, plus the improvement Δ_j in the value of u1 obtainable by fixing x_j at 1, exceeds this gap. Furthermore, since the costs c_j are integer but z_L, s_j and Δ_j may be fractional, x_j can be fixed at 0 if $\lceil z_L + s_j + \Delta_j \rceil \geq z_U$. Since $\Delta_j \leq c_j - s_j$, in order to avoid spending an unnecessary effort on computing Δ_j, we first check whether $\lceil z_L + c_j \rceil \geq z_U$, and only if this is the case do we calculate Δ_j.

The fixing of variables at 0 may reduce the size of some N_i to 1, i.e. create some rows with a single nonzero entry. Clearly, if this entry occurs in column k, x_k can be fixed at 1. Fixing x_k at 1 requires the removal ("flagging") of all rows $i \in M_k$ and the updating of the reduced costs. This in turn

may allow for some dual variables to be increased, and if this happens and as a result the integrality gap is again reduced, a new cycle of variable fixing can be started. Our routine which implements this cycle of variable fixing and associated updating of dual variables and reduced costs is called Recursive Fixing (RECURFIX).

```
procedure RECURFIX
input: c, A, cover x', z_U = cx', feasible u, z_L = ul
output: x_j's fixed (at 0 or 1), maximal u, improved z_U, z_L
begin
    z_U^0 := ∞, z_L^0 := 0, s^0 := c - uÃ {initialize}
    while z_U < z_U^0 or z_L > z_L^0 or s_j > s_j^0 for some j do
        z_U^0 := z_U, z_L^0 := z_L, s^0 := s;
        for all j ε N do
            if ⌈z_L + c_j⌉ ≥ z_U then {temporarily fix x_j at 1}
                ū := u, s̄ := s
                for i ε M_j do {update ū and s̄}
                    if ū_i > 0 then
                        for ℓ ε N_i do
                            if s̄_ℓ = 0 then mark rows i ε M_ℓ
                            endif
                            s̄_ℓ := s̄_ℓ + ū_i
                        endfor
                        ū_i = 0
                    endif
                endfor
                M := M\M_j
                call DUALMAX with u := ū {make ū maximal}
                Δ_j := Σ(ū_i - u_i : i ε M_j)
            if ⌈z_L + s_j + Δ_j⌉ ≥ z_U then
                fix x_j at 0;
                for i ε M_j do {update N_i}
                    N_i := N_i\{j}
                endfor
            endif
        endfor
```

```
        for all i ε M do
            if |N_i| = 1, N_i = {j}, then
                fix x_j at 1
                for i ε M_j do {update u and s}
                    if u_i > 0 then
                        for ℓ ε N_i do
                            if s_ℓ = 0 then
                                mark rows i ε M_ℓ
                            endif
                            s_ℓ := s_ℓ + u_i
                        endfor
                        u_i = 0
                    endif
                endfor
                M := M\M_j {"flag" rows in M_j}
            endif
        endfor
        call DUALMAX {make u maximal}
        F1 := {j ε N : x_j is fixed at 1 or x'_j = 1 and x_j is not fixed}
        complete F1 to a cover x
        if cx < z_U then
            z_U := cx
        endif
    endwhile
end
```

The routine DUALMAX called after fixing a variable at 1 and updating the reduced costs, serves the purpose of increasing those components of u for which this has been made possible by the updating (augmentation) of some reduced costs. This routine is also called on other occasions, which is the reason for stating it as a separate entity.

```
procedure DUALMAX
input: c, A, M, feasible u, M* {set of marked rows}
output: maximal u
```

```
begin
    for i ε M* do {make u maximal}
        if s_ℓ > 0 for all ℓ ε N_i then
            s_k := min(s_ℓ : ℓ ε N_i)
            u_i := u_i + s_k
            for ℓ ε N_i do {update reduced costs}
                s_ℓ := s_ℓ - s_k
            endfor
        endif
    endfor
end.
```

We count the cycles of the recursive fixing procedure by the number of variables that are fixed at 1. Of course, during a cycle several variables may be fixed at 0. If Z is the set of these variables and k is the index of the variable fixed at 1, it takes $O(n + q_Z)$ time to fix all those variables that can be fixed at 0 during on cycle, where q_Z is the number of nonzero entries in the columns indexed by Z. Further, it takes $O(m + q_k)$ time to fix x_k at 1 and update the reduced costs, with q_k denoting the number of nonzero entries in the rows covered by column k. Finally, DUALMAX is O(q). We conclude that the complexity of a cycle of RECURFIX is O(q).

7. Dynamic Subgradient Optimization

We are now almost ready to state the Dynamic Subgradient procedure in its entirety. Before we proceed with this, however, there are a couple of issues to be settled.

The first question has to do with the fact that if one of the primal heuristics finds an improvement in the upper bound z_U, this produces a sudden decrease in the gap $z_U - z_L$, and hence in the step length σ^t. If the recursive fixing procedure is successful, this decrease in σ^t is even sharper. Our experience has been that this leads to a premature stopping of the subgradient procedure. To counteract this effect, every time there is a change in the upper

bound z_U we readjust upwards the step length parameter λ in order to compensate for the decrease in $z_U - z_L$. To be specific, after every change in z_U we redefine λ as

$$\lambda := \min \left\{ 2, \max \left\{ \lambda, \sigma^{t-1} \frac{\|g(u^t)\|^2}{z_U - z_L} \right\} \right\}.$$

The effect of this is that, if the maximum of the two expressions in the inner brackets is attained for the second one, and the latter does not exceed 2, then $\sigma^t = \sigma^{t-1}$, i.e. the new step length is going to be the same as the one in the previous iteration. If, on the other hand, the maximum is attained for λ -- which can happen if the decrease in the gap $z_U - z_L$ is offset by a decrease in $\|g(u)\|^2$ -- then λ remains unchanged. It has been our experience that this readjustment of λ accomplishes what it is meant for, i.e. it eliminates the above mentioned premature stopping.

The second question is how often during the subgradient optimization procedures should the embedded primal heuristics be called. Here there are obvious tradeoffs, in that more frequent calls of the heuristics produce better solutions, but at a higher computational cost. We are using two versions of the Dynamic Subgradient procedure.

Version 1, stated below, calls ERCH only after the parameter λ has undergone at least two reductions, i.e. $\lambda < 1$, and only after those iterations that produce an improvement in the lower bound. This second criterion is motivated by our empirical observation that ERCH tends to have a markedly better performance when the dual solution it is given as input is of good quality.

Version 2 differs from Version 1 in that it calls an embedded primal heuristic at every iteration t (for $t \geq 5$), and it alternates between calling ERCH and ERCGH. Also, Version 2 reduces the value of λ less frequently than Version 1.

```
procedure DYNSGRAD
input:    c, A, M̄, z_U, z_L, u^1; λ, k, ε, δ, Ω
output:   {improved} z_L and z_U, x_j's fixed (at 0 or 1), x, u^t
begin
     α := 0 {initialize}
     while z_U > ⌈z_L⌉ do
          solve L(u^t) {find x(u^t)}
          if L(u^t) > z_L then
               z_L := L(u^t), α := 1
          endif
          if z_L unchanged for k iterations then
               λ := λ/2
          endif
          compute subgradient g(u^t)
          if α = 1 and λ < 1 then
               call ERCH {make x(u^t) into a cover x}
               if cx < z_U then
                    z_U := cx
                    call DUALFEAS {make u^t feasible}
                    call RCH {find new x and u}
                    z_U := min{z_U, cx}, z_L := max{z_L, u1}
                    call RECURFIX {try to fix variables}
                    update M̄ {find set of disjoint constraints}
                    define u^t from u
                    solve L(u^t) {find x(u^t)}
                    update z_L
                    compute subgradient g(u^t)
                    adjust λ
               endif
               α := 0
          endif
          if σ^t < ε or ‖g(u^t)‖ < δ or t > Ω or N_i = ø for some i ε M then
               stop
          endif
          compute step length σ^t
          for i ε M\M̄ do {update u_i}
               u_i^{t+1} := u_i^t + σ^t g_i(u^t)
          endfor
```

endwhile

end.

The following are typical values of the parameters we use with DYNSGRAD:

starting value for $\lambda = 2$

k (the number of iterations without improvement in z_L after which λ is halved) = 7 (Version 1), 30 (Version 2)

ε (threshold value for σ^t, stopping parameter) = 0.0001 at the root node of the search tree, 0.001 at subsequent nodes.

δ (threshold value for $\|g(u^t)\|$, stopping parameter) = 0.000001

Ω (threshold value for number of iterations, stopping parameter) = 1,800

There remains a last question to be addressed. The standard subgradient procedure, as stated in Section 3, is known to have certain convergence properties: if the upper bound z_U is "close enough" to the value of an optimal solution to $\max\{L(u) : u \geq 0\}$, then u^t converges to such an optimal solution. The question is, will the modified procedure still have this property? To answer this question, we note that between any two changes in the value of z_U, the procedure essentially reduces to the standard SUBGRAD: although ERCH may be called, its running has no consequences if the cover it finds does not improve on z_U. Further, the number of times z_U changes is bounded by $z_U - z_L$, i.e. is finite, since z_U is integer. Hence DYNSGRAD preserves the convergence properties of SUBGRAD. Further, since any interruption in the standard subgradient iterations is accompanied by the replacement of z_U with a value that is smaller, i.e. closer to $\max\{L(u) : u \geq 0\}$, the effect of these interruptions should be a speedup of convergence. Our computational experience bears this out.

As to the complexity of DYNSGRAD, the standard SUBGRAD is O(q); each of the routines ERCH, DUALFEAS and RCH is O(q); and each cycle of RECURFIX is O(q). hence the complexity of DYNSGRAD with a constant number of cycles of RECURFIX is

O(q). On the other hand, if the number of cycles of RECURFIX is allowed to increase with n, the complexity becomes O(nq). In practice, the number of cycles of RECURFIX is typically between 1 and 4 and we observed no increase in this number with the increase of n.

8. Branch and Bound

We now turn to the embedding of the Dynamic Subgradient Procedure into a branch and bound algorithm. We use a breadth first search strategy, i.e. we keep the active subproblems in a priority queue and examine them in order of increasing lower bound value. At the start, we apply primal and dual greedy heuristics to the initial problem P_0 in order to find feasible solutions x and u with associated bounds z_U and z_L, and try to fix at 0 or 1 as many variables as possible. At subsequent nodes of the search tree these steps are skipped.

We then construct the Lagrangean dual by choosing a maximal set of disjoint constraints, and call the Dynamic Subgradient Procedure. This is the main engine of our branch and bound algorithm, and it does one or several of the following things: improves the lower bound z_L; improves the upper bound z_U; fixes some variables at 0 or 1. When command is returned to the main program, the procedures DUALFEAS, RCH and RECURFIX are called in succession to make the dual solution feasible, to attempt to find a better cover, and to recursively fix as many variables as possible. The resulting partial cover is then completed to a full cover.

At this point, if the lower bound equals the upper bound, the current subproblem is discarded and a new one is chosen from the priority queue. Otherwise, we branch.

The branching rule used with the algorithm creates two new subproblems such that with a high probability there will be an improvement in the lower bound for both subproblems. This is accomplished by the following two rules, which

are applied in a hierarchical fashion, i.e. the second rule is used when the first one is not applicable.

Rule 1. (Column Branching) Let $N^0 := \{j \in N : s_j = 0\}$, and let J be the set of those $j \in N^0$ that cover at least one row i with $u_i > 0$, not covered by any other column in N^0. Such columns may or may not exist. If they don't, i.e. if $J = \emptyset$, rule 1 is not applicable. Otherwise choose for branching a column $k \in J$, and fix x_k at 0 on the left branch and at 1 on the right branch. As to the choice of k within J, we select $k \in J$ such that the set N_i for some $i \in M_k$ contains a maximal number of columns with zero reduced cost.

The rationale for this rule is as follows. On the left branch, where x_k is fixed at 0, the dual variable u_i corresponding to the row i such that $N_i \cap N^0 = \{k\}$, can be increased without changing the other components of u, which improves the lower bound. On the other hand, on the right branch, where x_k is fixed at 1, all the rows of M_k are eliminated (flagged) and for each $i \in M_k$ such that $u_i > 0$ the reduced costs of all $k \in N_i$ are increased. If some of these reduced costs were 0, the fact that they become positive may permit the increase of some components of u and thereby of the lower bound. Thus the choice of k within the set J by the criterion described above aims at having on the right branch as many 0 reduced costs augmented as possible.

Rule 2. (Row Branching) Let $r \in M$ be a row with a minimum $|N_i \cap N^0|$ among all those rows i such that some column $\ell \in N_i$ has positive reduced cost s_ℓ. Then on the left branch fix at 0 those x_j with $j \in N_r$ such that $s_j = 0$, and on the right branch replace the inequality indexed by r with the tighter inequality $\Sigma(x_j : j \in N_r \text{ and } s_j = 0) \geq 1$. On the left branch the dual variable u_r can be increased, and hence the lower bound strengthened, since the new set N_r will contain no column with 0 reduced cost. On the right branch such an increase is not guaranteed, but the tightening of the constraint indexed by r may produce it, and the choice of r is meant to maximize this tightening.

As soon as we create the two branches, we compute a lower bound z_L for each subproblem, by marking the appropriate rows and calling DUALFEAS. Subproblems for which $z_L \geq z_U$ are discarded, the others are added to the priority queue.

The procedures implementing the two branching rules follow. Let Q stand for the priority queue which contains the active subproblems P_i ordered according to nondecreasing lower bound z_L^i. The current subproblem is denoted P_t.

```
procedure BRANCH1
input: c, A, M̄, z_U, z_L, feasible u, N^0, Q, P_t
output: 0, 1 or 2 new subproblems
begin
    J := ∅ {initialize}
    for j ∈ N^0 do
        for i ∈ M_j do
            if u_i > 0 and |N_i ∩ N^0| = 1 then
                J := J ∪ {j}
                return
            endif
        endfor
        d_j := max{|N_i ∩ N^0| : i ∈ M_j, u_i > 0}
    endfor
    if J ≠ ∅ then
        d_k = max{d_j : j ∈ J}
        open subproblem P_t1
            fix x_k at 0
            compute z_L^1 {lower bound}
        end
        open subproblem P_t2
            fix x_j at 1
            compute z_L^2 {lower bound}
        end
        for α = 1,2 do
            if ⌈z_L^α⌉ ≥ z_U then discard P_tα
            else insert P_tα into Q
        endfor
```

```
    endif
end.

procedure BRANCH2
input: c, A, M̄, z_U, z_L, feasible u, s, N^0, Q, P_t
output: 0, 1 or 2 new subproblems
begin
    |N_r ∩ N^0| := min {|N_i ∩ N^0| : i ε M and N_i\N^0 ≠ ∅}
    open subproblem P_t1
        for j ε N_r do
            if s_j = 0 then
                fix x_j at 0
            endif
        endfor
        compute z_L^1 {lower bound}
    end
    open subproblem P_t2
        for j ε N_r do
            if s_j > 0 then
                change a_rj = 1 to a_rj = 0
            endif
        endfor
        compute z_L^2 {lower bound}
    end
    for α = 1,2 do
        if ⌈z_L^α⌉ ≥ z_U then discard P_tα
        else insert P_tα into Q
    endfor
end.
```

We are now ready to give an overview of the branch and bound algorithm as a whole. We denote by Q the priority queue of active subproblems, and by P_0 the starting subproblem.

```
procedure BR&BND
input: c, A
output: x {optimal cover}
begin
     z_U := ∞, z_L := 0, Q := {P_0} {initialize}
     while Q ≠ ∅ do
          remove from Q the next subproblem P_i
          if i = 0 then
               use primal and dual heuristics to find x, u, z_U and z_L
               call RECURFIX {try to reduce problem size}
          endif
          construct Lagrangean dual {find disjoint constraint set M̄}
          call DYNSGRAD {find new u, z_L, x, z_U}
          call DUALFEAS {make u feasible}
          call RCH {find new x}
          call RECURFIX {try to reduce problem size}
          complete x to a cover
          if ⌈z_L⌉ ≥ z_U then discard P_i and return
          else
               call BRANCH1 {try to do column branching}
               if no branching variable found then
                    call BRANCH2 {do row branching}
               endif
          endif
     endwhile
end
```

We use two versions of BR&BND; they differ among each other in that Version 1 of BR&BND uses Version 1 of DYNSGRD, whereas Version 2 uses Version 2 of DYNSGRD.

9. Computational Experience

Our procedures were tested on 86 large, mostly sparse set covering problems. Table 1 describes the first 60 test problems, which were randomly generated. The first 25 of these problems are from Balas and Ho [80], the next 10

Table 1. Randomly Generated Data Sets
$c_{ij} \in [1,100]$

Problem Set	Number of Problems	Number of: Rows	Columns	Density	Cost Range
BH4	10	200	1000	2%	[1,100]
BH5	10	200	2000	2%	[1,100]
BH6	5	200	1000	5%	[1,100]
BC1	5	200	1000	10%	[1,100]
BC2	5	300	5000	2%	[1,100]
BeA	5	300	3000	2%	[1,100]
BeB	5	300	3000	5%	[1,100]
BeC	5	400	4000	2%	[1,100]
BeD	5	400	4000	5%	[1,100]
BeE	5	50	500	20%	[1,1]*

*unit costs

Table 2. Real World Data Sets

Problem	Number of Rows	Number of Columns	Density	Cost Range
AA02	106	9444	4.05%	[91,3619]
AA03	106	8661	4.05%	
AA04	106	8002	4.05%	
AA05	105	7435	4.05%	
AA06	105	6951	4.11%	
AA07	105	6526	4.11%	
AA08	106	6142	4.05%	
AA09	104	5810	4.16%	
AA10	106	5511	4.06%	
AA11	71	4413	2.53%	[35,2966]
AA12	272	4208	2.52%	
AA13	265	4025	2.60%	
AA14	266	3868	2.50%	
AA15	267	3701	2.58%	
AA16	265	3558	2.63%	
AA17	264	3425	2.61%	
AA18	271	3314	2.55%	
AA19	263	3202	2.63%	
AA20	269	3095	2.58%	
AA21	55	739	11.3%	
AA22	62	908	11.0%	
AA23	66	4999	13.6%	
AA24	50	2707	16.8%	
AA25	55	4999	15.8%	
AA26	61	4522	13.3%	
AA27	60	3954	13.2%	
AA28	56	4999	15.3%	
AA29	46	1681	17.7%	
AA30	41	3334	21.3%	
BUS1	454	2241	1.89%	
BUS2	681	9524	0.51%	

were generated for the purposes of this study, and the last 25 are from Beasley [87,90].

Table 2 describes the remaining 31 test problems, which have a real-world origin. All but the last two of them are airline crew scheduling problems kindly provided to us by American Airlines. The last two are bus driver scheduling problems originating with a Canadian consulting firm.

We first discuss our computational experience with DYNSGD as a stand-alone heuristic.

Table 3 compares the performance of various primal heuristics with that of the Dynamic Subgradient Procedure, in terms of the quality of the solutions they produce. The comparison is made on the 35 problems of the sets BH4, BH5, BH6, BC1 and BC2. The heuristics compared are the straight greedy with the choice criterion c_j/k_j, the greedy run with 9 different criterion functions $f(c_j,k_j)$ followed by the choice of the best outcome, the randomized greedy run 30 times, the reduced cost heuristic (RCH) run with the dual vector provided by SUBGRAD, and finally the two versions of DYNSGRAD. The Dynamic Subgradient Procedure found optimal solutions for 17 (Version 1), respectively for 21 (Version 2) of the 35 test problems. In not a single problem did any of the other heuristics find a better solution than Version 2 of DYNSGRAD, and only in 5 out of 35 problems did any of the other heuristics find a better solution than Version 1 of DYNSGRAD.

Table 4 compares the two versions of DYNSGRAD with the randomized greedy and with the reduced cost heuristic run with the dual vector found by SUBGRAD, on the 31 crew scheduling problems of the sets AA2-AA10, AA11-AA20, AA21-AA30, and BUS1-BUS2. DYNSGRAD found optimal solutions for 6 (Version 1), respectively 10 (Version 2) of the first 20 problems, and for all but one of the remaining problems. The solution found by DYNSGRAD was in all cases better than the one found by the randomized greedy, and in almost all the cases better than or equal

Table 3. Value of covers found by various procedures

Problem	Greedy with c_j/k_j	Greedy best of 9 $f(c_j,k_j)$	Randomized Greedy (30 runs)	RCH after SUBGRAD	DYNSGRAD Version 1	DYNSGRAD Version 2	Optimum
BH4.1	434	434	432	430	429*	429*	429
BH4.2	529	529	524	522	512*	512*	512
BH4.3	537	537	532	519	516*	516*	516
BH4.4	506	506	504	496	496	494*	494
BH4.5	518	518	518	512*	512*	512*	512
BH4.6	594	582	573	570	561	560*	560
BH4.7	447	447	445	430*	430*	430*	430
BH4.8	525	509	508	498	492*	492*	482
BH4.9	664	664	666	661	641*	641*	641
BH4.10	528	523	521	521	514*	514*	514
BH5.1	269	269	258	268	259	254	253
BH5.2	330	318	312	324	311	307	302
BH5.3	232	230	229	226	226*	226*	226
BH5.4	250	247	250	261	244	243	242
BH5.5	214	214	217	216	211*	211*	211
BH5.6	226	216	221	213	213*	213*	213
BH5.7	306	301	304	299	295	293*	293
BH5.8	311	305	307	292	289*	288*	288
BH5.9	292	285	281	282	279*	279*	279
BH5.10	277	275	274	265	265*	265*	265
BH6.1	142	142	142	167	142	140	138
BH6.2	156	156	152	157	156	147	146
BH6.3	157	151	148	149	145*	145*	145
BH6.4	140	135	132	135	132	131*	131
BH6.5	186	181	176	181	170	163	161
BC1.1	50		49	54	48	47*	47
BC1.2	62		53	65	57	53*	53
BC1.3	58		58	58	58	56	55
BC1.4	43		43	42	41*	42	41
BC1.5	68		64	91	66	64	63
BC2.1	165		161	156	157	151	148
BC2.2	147		141	144	138*	141	138
BC2.3	156		154	158	152	151	149
BC2.4	176		177	173	173	168	167
BC2.5	149		149	146	149	143	140

*Optimal value

Table 4. Value of covers found by four procedures

Problem	Randomized Greedy (30 runs)	RCH after SUBGRAD	DYNSGRAD		Optimum
			Version 1	Version 2	
AA2	34,812	31,753*	31,753*	31,753*	31,753
AA3	35,794	33,852	33,157	33,157	33,155
AA4	36,410	35,312	35,092	34,573*	34,573
AA5	32,616	31,623	32,144	31,623*	31,623
AA6	38,070	38,422	37,610	37,464*	37,464
AA7	37,479	35,545*	35,545*	35,545*	35,545
AA8	35,782	35,793*	34,731*	34,731*	34,731
AA9	36,230	34,295*	34,295*	34,295*	34,295
AA10	37,181	35,332*	35,332*	35,332*	35,332
AA11	38,576	35,614	35,478	35,478	35,384
AA12	33,416	31,266	30,859	30,815	30,809
AA13	36,645	33,211*	33,211*	33,211*	33,211
AA14	35,369	33,702	33,279	33,222	33,219
AA15	36,964	34,999	34,667	34,510	34,409
AA16	35,801	33,968	32,858	32,858	32,752
AA17	36,405	31,737	31,737	31,717	31,612
AA18	39,663	38,296	36,850	36,866	36,782
AA19	36,067	33,012	32,593	32,317*	32,317
AA20	36,791	36,175	34,994	35,160	34,912
AA21	4,518	4,338*	4,338*		4,338
AA22	4,791	4,532*	4,532*		4,532
AA23	4,370	4,200*	4,200*		4,200
AA24	3,135	3,045*	3,045*		3,045
AA25	3,944	4,529	3,661*		3,661
AA26	4,095	4,242	4,071*		4,063
AA27	4,154	4,031	3,871*		3,871
AA28	3,598	3,829	3,571		3,530
AA29	3,066	2,870*	2,870*		2,870
AA30	2,842	2,701*	2,701*		2,701
BUS1			27,947	27,947	27,947
BUS2			68,732	67,868	67,760

*Optimal value

to the solution found by the reduced cost heuristic applied after the standard subgradient.

Table 5 compares the running times of the above heuristics. Version 1 of DYNSGRAD took longer than greedy or RCH (even if the SGD time needed to find the dual solution that RCH uses is added to its computing time). On the first four problem sets it took less time than the randomized greedy, whereas on the last two sets it took longer. In view of the superior quality of the solutions that it finds, we must conclude that DYNSGRAD (Version 1) is to be preferred to the other heuristics except in those cases where speed is decidedly more important than solution quality. Version 2 in most cases took considerably longer than the randomized greedy, but it obtained considerably better solutions than the latter.

Finally, Table 6 compares the performance of the two versions of DYNSGRAD with that of Beasley's Lagrangean heuristic on the problem sets BH4, 5, 6 and BeA, B, C, D, E. The results for the first two Beasley columns are taken from Beasley [90], except for problem set E (not treated in Beasley [90]), for which the results are from Beasley [87].

It is hard to make exact comparisons between computing times on different computers. Nevertheless, we attempted to give an approximate idea of the relative magnitudes of the times reported. To do so, we used the report of Dongarra [92] (Argonne National Laboratory), which compares the speed of several hundred computers on the task of solving systems of linear equations. The column "TPP" (Toward Peak Performance) of Table 1 of Dongarra's report contains the number of Megaflops obtained as a result of programming efforts made to maximize the performance of each type of computer. The corresponding entries for the Cray 1S and the VAX 6420 are 110 and 3, respectively. In Table 6 (and later in Table 7) we use this ratio (110:3 = 36.67) to express the computing times on

Table 5. Running Times (VAX 8650, CPU millisec.)

Problem	Reading the Data	Greedy with c_j/k_j	Randomized greedy (30 runs)	SUBGRAD	RCH after SUBGRAD	DYNSGRAD Version 1	DYNSGRAD Version 2
BH5.1	680	890	18,630	7,400	140	11,000	58,300
BH5.2	630	850	19,090	10,400	130	12,700	101,200
BH5.3	620	790	18,150	4,900	150	3,400	5,900
BH5.4	590	790	18,260	8,000	130	11,200	60,800
BH5.5	580	840	18,950	11,600	140	20,500	12,000
BH5.6	590	810	17,960	8,500	130	4,500	7,600
BH5.7	630	910	19,730	8,200	130	11,300	39,800
BH5.8	630	910	18,930	8,800	110	11,000	22,600
BH5.9	580	790	17,750	7,900	120	4,500	8,900
BH5.10	690	850	18,490	5,400	130	4,600	6,300
BH6.1	750	320	12,950	7,100	130	8,300	47,700
BH6.2	640	320	12,710	7,600	140	8,700	47,600
BH6.3	640	340	12,980	5,500	130	9,100	60,200
BH6.4	660	310	13,260	5,700	120	10,500	44,200
BH6.5	670	310	12,980	5,400	140	8,300	53,600
BC1.1	1230	290	21,000	7,700	250	7,200	21,100
BC1.2	1260	310	20,920	10,700	300	9,000	40,500
BC1.3	1210	330	20,970	8,700	290	7,600	34,300
BC1.4	1150	340	21,200	8,300	260	7,200	29,600
BC1.5	1180	310	20,560	9,300	250	11,000	43,100
BC2.1	2380	2650	65,160	28,900	510	39,600	261,900
BC2.2	2390	2470	63,950	26,200	620	33,500	269,000
BC2.3	2370	2780	66,000	27,400	470	28,800	265,300
BC2.4	2300	2910	66,250	26,000	480	33,900	277,000
BC2.5	2290	2680	65,210	28,300	490	32,000	295,400
AA11	2500	1890	51,800	29,800	570	89,500	390,000
AA12	2140	1710	47,530	33,500	540	56,700	264,600
AA13	1980	1650	47,270	19,800	540	11,700	107,100
AA14	2040	1590	46,440	28,000	490	125,300	470,000
AA15	2010	1690	44,650	31,400	490	76,000	218,400
AA16	2790	1510	43,470	26,800	380	119,700	284,700
AA17	2410	1530	41,650	17,300	390	49,800	243,200
AA18	2300	1590	42,240	27,700	360	66,700	336,300
AA19	1910	1370	38,580	26,400	440	34,500	140,000
AA20	2250	1300	37,440	24,300	340	45,800	174,400

Table 6. The Two Versions of DYNSGRAD Compared to Beasley's Lagrangean Heuristic

Problem	DYNSGRAD				Beasley			Optimal Value
	Version 1		Version 2			Time		
	Value	Time (VAX6420)	Value	Time (VAX6420)	Value	Cray 1S sec	Equivalent VAX6420 sec (conversion factor† 110/3)	
BH4.1	429*	3.6	429*	4.8	429*	2.16	79.2	429
BH4.2	512*	3.2	512*	6.9	512*	2.18	79.9	512
BH4.3	516*	2.5	516*	5.9	516*	1.34	49.13	516
BH4.4	496	4.3	494*	9.8	495	2.39	87.6	494
BH4.5	512*	2.8	512*	3.2	512*	1.38	50.6	512
BH4.6	561	9.3	560*	40.5	561	3.11	114.0	560
BH4.7	430*	2.5	430*	5.8	430*	1.81	66.4	430
BH4.8	492*	9.2	492*	35.3	493	2.25	82.5	492
BH4.9	641*	8.0	641*	50.1	641*	4.04	148.13	641
BH4.10	514*	3.5	514*	6.3	514*	2.34	85.8	514
BH5.1	259	11.0	254	58.3	255	3.42	125.4	253
BH5.2	311	12.7	307	101.2	304	4.11	150.7	302
BH5.3	226*	3.4	226*	5.9	226*	1.98	72.6	226
BH5.4	244	11.2	243	60.8	242*	2.26	82.9	242
BH5.5	211*	20.5	211*	12.0	211*	1.42	52.1	211
BH5.6	213*	4.5	213*	7.6	213*	2.23	81.8	213
BH5.7	295	11.3	293*	39.8	294	3.55	130.2	293
BH5.8	289	11.0	288*	22.6	288*	4.06	148.9	288
BH5.9	279*	4.5	279*	8.9	279*	3.09	113.3	279
BH5.10	265*	4.6	265*	6.3	265*	1.93	70.8	265
BH6.1	142	8.3	140*	47.7	141*	3.30	121.0	138
BH6.2	156	8.7	147	47.6	146*	2.84	104.13	146
BH6.3	145*	9.1	145*	60.2	145*	2.94	107.8	145
BH6.4	132	10.5	131*	44.2	131*	2.02	74.1	131
BH6.5	170	8.3	163	53.6	162	2.61	95.7	161

†Crude approximation; see text for explanation
*Optimal value

Table 6 (Cont-d)

Problem	DYNSGRAD				Beasley			Optimal Value
	Version 1		Version 2			Time		
	Value	Time (VAX6420)	Value	Time (VAX6420)	Value	Cray 1S sec	Equivalent VAX6420 sec (conversion factor† 110/3)	
BeA.1	261	27.4	258	177.4	255	7.08	259.6	253
BeA.2	257	25.0	254	186.1	256	8.69	318.6	252
BeA.3	243	20.7	237	130.0	234	5.52	202.4	232
BeA.4	241	19.1	235	165.1	235	6.58	241.3	234
BeA.5	237	23.0	236*	119.1	237	3.73	136.8	236
BeB.1	72	24.2	69*	131.9	70	5.58	204.6	69
BeB.2	78	27.2	76*	131.7	77	8.02	294.1	76
BeB.3	81	26.9	81	159.6	80*	5.00	183.3	80
BeB.4	83	26.1	79*	131.6	80	7.26	266.2	79
BeB.5	75	25.5	72*	148.0	72*	5.11	187.4	72
BeC.1	235	39.7	230	528.0	230	8.33	305.4	227
BeC.2	224	36.4	220	255.2	223	12.92	473.7	219
BeC.3	249	44.0	248	280.4	251	14.75	540.8	243
BeC.4	233	35.2	224	309.5	224	12.45	456.5	219
BeC.5	219	32.5	217	293.8	217	7.84	287.5	215
BeD.1	64	45.0	61	166.4	61	8.03	294.4	60
BeD.2	70	46.1	67	211.9	68	10.35	379.5	66
BeD.3	76	56.7	74	214.6	75	10.97	402.2	72
BeD.4	67	40.5	63	180.7	64	7.17	262.9	62
BeD.5	61*	46.9	61	164.5	62	7.22	264.7	61
BeE.1	5*	12.4	5*	32.4	5*	18.3	671.0	5
BeE.2	5*	13.5	5*	42.4	5*	21.9	803.0	5
BeE.3	5*	14.9	5*	36.9	5*	23.5	861.7	5
BeE.4	6	3.4	5*	32.7	5*	19.5	715.0	5
BeE.5	5*	16.8	5*	35.8	5*	19.9	729.7	5

†Crude approximation; see text for explanation
*Optimal value

the Cray 1S in equivalent VAX 6420 times. This should be viewed as a very crude approximation.

In terms of solution quality, Beasley and Version 2 of DYNSGRAD are both clearly superior to Version 1 of DYNSGRAD, which in turn is about 7-15 times faster than Version 2, and about 10-20 times faster than the Beasley code. Version 2 of DYNSGRAD finds better solutions than Beasley's heuristic 17 times, while Beasley's procedure finds better solutions 7 times. The computing effort involved in Version 2 of DYNSGRAD is in most cases between 1/3 and 2/3 of the effort involved in Beasley's code, except for the (dense) problems of class E, where the comparison is made with Beasley [87] rather than [90] and where the ratio is about 1 to 20.

Next we turn to our computational experience with the branch and bound algorithm as a whole. As mentioned in section 8, Version 1 and Version 2 of BR&BND differ only in the version of DYNSGRAD that they use.

Table 7 compares the two versions of our branch and bound algorithm with the branch and bound algorithms of Fisher and Kedia [90], Beasley [87] and Beasley and Jornsten [92], on the problem sets on which all four codes have been run (BH4, 5 and 6); while Table 8 compares them with Beasley [87] and Beasley and Jornsten [92] on the problems sets BeA, B, C, D and E. The data for the Fisher-Kedia [90], Beasley [87] and Beasley-Jornsten [92] algorithms come from the papers cited, except for the columns titled "Equivalent VAX 6420 time," where we have used the conversion factors of 110/3 between the computing speed of the Cray 1S and that of the VAX 6420, and (2 x 184)/3 between that of the Cray X-MP/28 and the VAX 6420 taken from Dongarra [92] as explained above. Again, we remind the reader of the very crude nature of this approximation.

All the comparisons show that our branch and bound algorithm represents a clear-out progress relative to the previous state of the art.

Table 7. Comparison of Branch and Bound Algorithms

Problem	Fisher-Keida [90]		Beasley [87]		Beasley-Jornsten [92]		Balas - Carrera Version 1		Balas - Carrera Version 2	
	Search Tree Nodes	Time (Dec 10 sec)	Search Tree Nodes	Time (Cray 1S sec)	Search Tree Nodes	Time (Cray X-MP/28)	Search Tree Nodes	Time (VAX 6420 sec)	Search Tree Nodes	Time (VAX 6420 sec)
BH4.1	1	8.7	1	2.3	1	0.92	1	3.6	1	4.8
BH4.2	1	10.8	1	2.7	1	0.89	1	3.2	1	6.9
BH4.3	1	8.9	1	2.5	1	0.90	1	2.5	1	5.9
BH4.4	1	41.1	9	3.8	1	1.47	3	5.8	1	9.8
BH4.5	1	7.2	1	1.7	1	0.89	1	2.8	1	3.2
BH4.6	22	75.8	5	5.9	1	2.30	3	12.3	2	42.3
BH4.7	5	10.1	1	2.2	1	0.95	1	2.5	1	5.8
BH4.8	46	10.0	11	7.6	1	2.07	3	13.6	1	35.3
BH4.9	17	51.4	1	6.3	1	2.26	1	8.0	1	50.1
BH4.10	13	12.5	1	1.8	1	0.97	1	3.5	1	6.3
BH5.1	33	154.4	21	13.3	1	4.38	6	28.9	7	90.5
BH5.2	52	174.1	95	21.1	12	6.05	3	21.2	5	133.3
BH5.3	1	13.1	1	2.7	1	1.17	1	3.4	1	5.9
BH5.4	48	158.3	19	8.4	1	1.89	3	16.1	2	67.5
BH5.5	9	25.9	1	2.5	1	1.24	1	20.5	1	12.0
BH5.6	6	15.0	1	2.4	1	1.32	1	4.5	1	7.6
BH5.7	25	76.5	7	5.8	6	2.84	3	13.9	1	39.8
BH5.8	40	108.7	1	5.9	1	2.29	3	14.8	1	22.6
BH5.9	43	30.0	1	2.9	1	1.51	1	4.5	1	8.9
BH5.10	1	14.8	1	2.9	1	1.21	1	4.6	1	6.3
BH6.1	439	1,242.2	195	25.4	42	7.81	77	200.9	15	139.2
BH6.2	451	1,089.6	171	27.6	68	6.99	46	123.7	23	198.1
BH6.3	136	569.8	15	11.1	34	5.16	7	22.1	3	72.2
BH6.4	26	68.1	3	6.2	6	2.79	4	14.8	1	44.7
BH6.5	369	1,427.7	271	35.7	130	9.17	53	140.8	20	193.4

Table 8. Further Comparison of Branch and Bound Algorithms

Problem	Beasley [87]			Beasley-Jorsten [92]			Balas - Carrera		Balas - Carrera	
		Time			Time		Version 1		Version 2	
	Search Tree Nodes	Cray 1S sec	Equivalent VAX 6420 sec (conversion factor*: 110/3)	Search Tree Nodes	Cray X-MP/28 sec	Equivalent VAX 6420 sec (conversion factor*: (2 × 184)/3)	Search Tree Nodes	Time VAX 6420 sec	Search Tree Nodes	Time VAX 6420 sec
BeA.1	515	77.6	1,845.3	373	28.26	3,466.6	95	797.6	74	1,455.4
BeA.2	809	104.9	3,846.3	357	27.78	3.407.7	28	218.2	25	597.6
BeA.3	857	110.8	4,062.7	377	24.12	2,958.7	41	302.9	29	637.7
BeA.4	63	31.1	1,140.3	17	6.33	776.5	10	83.2	5	235.2
BeA.5	79	28.4	1,041.3	15	4.74	581.4	3	34.9	1	119.1
BeB.1	411	82.1	3,010.3	137	20.66	2,534.3	81	478.5	24	492.1
BeB.2	3,273	396.8	14,549.3	1,655	78.93	9,682.1	939	5,748.7	483	8,406.0
BeB.3	1,283	149.6	5,485.3	671	38.74	4,752.1	288	1,579.5	181	3,378.8
BeB.4	4,381	654.0	20,680.0	3,157	125.27	15,366.4	1,508	9,438.8	974	17,128.2
BeB.5	64	90.5	3,318.3	355	22.99	2,809.1	69	430.2	47	808.4
BeC.1	1,265	203.4	7,458.0	549	51.92	6,368.9	34	394.0	22	1,184.4
BeC.2	275	113.1	4,147.0	619	62.10	7,617.6	145	1,620.4	104	3,604.0
BeC.3	15,757	2,042.5	74,891.7	5,957	297.54	36,498.2	291	3,035.4	458	14,661.5
BeC.4	493	130.0	4,766.7	515	53.15	6,519.7	116	1,349.1	55	2,040.3
BeC.5	375	107.6	3,945.3	1,001	61.67	7,564.9	91	916.8	38	1,294.1
BeD.1	3,393	578.0	2,152.3	1,015	67.59	8,291.0	337	2,726.5	328	8,166.5
BeD.2	6,739	1,056.2	38,727.3	7,179	335.92	41,206.2	2,771	24,195.7	1,498	35,106.1
BeD.3	19,707	2,971.4	108,951.3	10,455	471.95	57,892.5	5,120	50,466.6	2,219	56,502.4
BeD.4	12,389	1,947.9	71,423.0	8,687	376.78	46,218.3	2,931	28,266.1	1,631	40,039.8
BeD.5	147	76.1	2,790.3	63	22.72	2,787.0	18	170.9	14	464.0
BeE.1	69	28.1	1,030.3	39	19.06	2,338.0	9	45.5	6	60.0
BeE.2	145	46.5	1,705.0	109	24.54	3,010.2	21	194.5	24	271.5
BeE.3	231	62.9	2,306.3	131	28.43	3,487.4	28	410.6	33	553.3
BeE.4	129	38.7	1,419.0	59	17.20	2,109.9	64	440.0	22	181.5
BeE.5	167	47.8	1,752.7	127	21.89	2,685.2	26	300.4	23	265.8

10. Conclusions

We have presented a new branch and bound algorithm for set covering, whose main ingredient is an integrated upper bounding/lower bounding procedure applied to a Lagrangean dual. This Dynamic Subgradient Procedure (DYNSGRAD) combines the standard subgradient method with primal and dual heuristics that change the Lagrange multipliers, fix variables and restate the Lagrangean itself during the procedure. Extensive computational experience on randomly generated as well as real-world set covering problems show superior performance of DYNSGRAD both as a stand-alone heuristic, and as part of a branch and bound algorithm.

Experiments are under way with another version of DYNSGRAD and BR&BND, in which the Lagrangean dual incorporates cutting planes that are guaranteed to improve the lower bound. These, however, will be discussed in another paper.

REFERENCES

E. Balas and A. Ho (1980), "Set Covering Algorithms Using Cutting Planes, Heuristics and Subgradient Optimization: A Computational Study." *Mathematical Programming, 12,* pp. 37-60.

J.E. Beasley (1987), "An Algorithm for Set Covering Problems." *European Journal of Operational Research, 31,* pp. 85-93.

J.E. Beasley (1990), "A Lagrangean Heuristic for Set Covering Problems." *Naval Research Logistics, 37,* pp. 151-164.

J.E. Beasley and K. Jornsten (1992), "Enhancing an Algorithm for Set Covering Problems." *European Journal of Operational Research, 58,* pp. 293-300.

M.C. Carrera (1984), "Computational Study of Set Covering Heuristics." Master's thesis, Department of Mathematics, Carnegie Mellon University.

J. Dongarra (1992), "Performance of Various Computers Using Standard Linear Equations Software." Mathematical Sciences Section, Oak Ridge National Laboratory and Computer Science Department, University of Tennessee.

J. Etcheberry (1977), "The Set Covering Problem: A New Implicit Enumeration Algorithm." *Operations Research, 25,* pp. 760-772.

M.L. Fisher and P. Kedia (1990), "Optimal Solution of Set Covering Problems Using Dual Heuristics." *Management Science, 36,* pp. 674-688.

N.G. Hall and D.S. Hochbaum (1985), "A Fast Approximation Algorithm for the Multicovering Problem." Technical Report, Ohio State University.

F. Harche and G.L. Thompson (1989), "The Column Subtraction Algorithm: An Exact Method for Solving Weighted Set Covering Problems." MSRR No. 548, GSIA, Carnegie Mellon University.

P. Nobili and A. Sassano (1992), "A Separation Routine for the Set Covering Polytope." In E. Balas, G. Cornuejols and R. Kannan (editors), Integer Programming and Combinatorial Optimization, Proceedings of the 2nd IPCO Conference, Carnegie Mellon University Press, 1992.

F.J. Vasko and G.R. Wilson (1984), "An Efficient Heuristic for Large Set Covering Problems." *Naval Research Logistics Quarterly, 31,* pp. 163-171.

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER MSRR-568(R)	2. GOVT ACCESSION NO
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle) A DYNAMIC SUBGRADIENT-BASED BRANCH AND BOUND PROCEDURE FOR SET COVERING	5. TYPE OF REPORT & PERIOD COVERED Technical report; October 1991, revised May 1992
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(S) Egon Balas Maria C. Carrera	8. CONTRACT OR GRANT NUMBER(S) DDM-8901495 N00014-89-J1063
9. PERFORMING ORGANIZATION NAME AND ADDRESS Graduate School of Industrial Administration Carnegie Mellon University Pittsburgh, PA 15213-3890	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Personnel and Training Research Programs Office of Naval Research (Code 434) Arlington, VA 22217	12. REPORT DATE October 1991; revised May 1992
	13. NUMBER OF PAGES 28
14. MONITORING AGENCY NAME & ADDRESS (If different from Controlling Office)	15. SECURITY CLASS (of this report)
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Set Covering
Subgradient Optimization
Heuristics
Branch and Bound

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

We discuss a branch and bound algorithm for set covering, whose centerpiece is an integrated upper bounding/lower bounding procedure called dynamic subgradient optimization (DYNSGRAD), that is applied to a Lagrangean dual problem at every node of the search tree. Extensive experimentation has shown DYNSGRAD to represent a significant improvement over currently used techniques.